Pipelining

Reducing Instruction Execution Time

Dr. Javier Navaridas
javier.navaridas@manchester.ac.uk
Overview and Learning Outcomes

• Deepen the understanding on how modern processors work

• Learn how pipelining can improve processors performance and efficiency

• Being aware of the problems arising from using pipelined processors

• Understanding instruction dependencies
The Fetch-Execute Cycle

- As explained in COMP15111 Instruction execution is a simple repetitive cycle:

```
LDR R0, x
LDR R1, y
ADD R2, R1, R0
STR R2, Z
...
```

Fetch Instruction

Execute Instruction
Fetch-Execute Detail

The two parts of the cycle can be further subdivided

• Fetch
  – Get instruction from memory (IF)
  – Decode instruction & select registers (ID)

• Execute
  – *Perform operation or calculate address (EX)*
  – *Access an operand in data memory (MEM)*
  – *Write result to a register (WB)*

We have designed the ‘worst case’ data path
  – It works for all instructions
### Processor Detail

#### IF
- Instruction Fetch

#### ID
- Instruction Decode

#### EX
- Execute Instruction

#### MEM
- Access Memory

#### WB
- Write Back

<table>
<thead>
<tr>
<th>Cycle i</th>
<th>Instruction</th>
<th>ID Action</th>
<th>EX Action</th>
<th>MEM Action</th>
<th>WB Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDR R0, x</td>
<td>Select register (PC)</td>
<td>Compute address x</td>
<td>Get value from [x]</td>
<td>Write in R0</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cycle i+1</th>
<th>Instruction</th>
<th>ID Action</th>
<th>EX Action</th>
<th>MEM Action</th>
<th>WB Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD R2,R1,R0</td>
<td>Select registers (R0, R1)</td>
<td>Add R0 &amp; R1</td>
<td>Do nothing</td>
<td>Write in R2</td>
<td></td>
</tr>
</tbody>
</table>
Cycles of Operation

• Most logic circuits are driven by a clock
• In its simplest form one instruction would take one clock cycle (single-cycle processor)
• This is assuming that getting the instruction and accessing data memory can each be done in a $1/5^{th}$ of a cycle (i.e. a cache hit)
• For this part we will assume a perfect cache replacement strategy
Logic to do this

- Each stage will do its work and pass to the next
- Each block is only doing useful work once every $\frac{1}{5}$th of a cycle
### Application Execution

<table>
<thead>
<tr>
<th>Clock Cycle</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>LDR</strong></td>
<td>IF ID EX MEM WB</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>LDR</strong></td>
<td></td>
<td>IF ID EX MEM WB</td>
<td></td>
</tr>
<tr>
<td><strong>ADD</strong></td>
<td></td>
<td></td>
<td>IF ID EX MEM WB</td>
</tr>
</tbody>
</table>

- Can we do it any better?
  - Increase utilization
  - Accelerate execution
Insert Buffers Between Stages

- Instead of direct connection between stages – use extra buffers to hold state
- Clock buffers once per cycle
In a pipeline processor

• Just like a car production line
• We still can execute one instruction every cycle
• But now clock frequency is increased by 5x
• 5x faster execution!

Clock Cycle

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDR</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDR</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADD</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Benefits of Pipelining

Without Pipelining

<table>
<thead>
<tr>
<th></th>
<th>Clock Cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>LDR</td>
<td>IF ID EX MEM WB</td>
</tr>
<tr>
<td>LDR</td>
<td></td>
</tr>
<tr>
<td>ADD</td>
<td></td>
</tr>
</tbody>
</table>

With Pipelining

<table>
<thead>
<tr>
<th></th>
<th>Clock Cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>LDR</td>
<td>IF</td>
</tr>
<tr>
<td>LDR</td>
<td>IF</td>
</tr>
<tr>
<td>ADD</td>
<td>IF</td>
</tr>
</tbody>
</table>
Why 5 Stages?

• Simply because early pipelined processors determined that dividing into these 5 stages of roughly equal complexity was appropriate

• Some recent processors have used more than 30 pipeline stages

• We will consider 5 for simplicity at the moment
Imagine we have a non-pipelined processor running at 10MHz and want to run a program with 1000 instructions.

a) How much time would it take to execute the program?
Pipelining Example

Assuming ideal conditions (perfect pipelining and no hazards), how much time would it take to execute the same program in:

b) A 10-stage pipeline?

c) A 100-stage pipeline?
Pipelining Example

Looking at those results, it seems clear that increasing pipeline should increase the execution speed of a processor. Why do you think that processor designers (see Intel, below) have not only stopped increasing pipeline length but, in fact, reduced it?

- **Pentium III – Coppermine (1999)** – 10-stage pipeline
- **Pentium Prescott (2004)** – 31-stage pipeline
- **Core i7 9xx – Bloomfield (2008)** – 24-stage pipeline
- **Core i7 5Yxx – Broadwell (2014)** – 19-stage pipeline
Limits to Pipeline Scalability

• Higher freq. => more power consumed

• More stages
  – more extra hardware
  – more complex design (forwarding?)
  – more difficult to split into uniform size chunks
  – loading time of the registers limits cycle period

• Hazards (control and data)
  – A longer datapath means higher probability of hazards occurring and worse penalties when they happen
Control Hazards
The Control Transfer Problem

• Instructions are normally fetched sequentially (i.e. just incrementing the PC)

• What if we fetch a branch?
  – We only know it is a branch when we decode it in the second stage of the pipeline
  – By that time we are already fetching the next instruction in serial order
A Pipeline ‘Bubble’

We must mark Inst 5 as unwanted and ignore it as it goes down the pipeline. But we have wasted a cycle.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inst 3</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B n</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Inst 5</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Inst n</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

We know it is a branch here. Inst 5 is already fetched.
Conditional Branches

- It gets worse!
- Suppose we have a conditional branch
- It is possible that we might not be able to determine the branch outcome until the execute (3rd) stage
- We would then have 2 ‘bubbles’
- We can often avoid this by reading registers during the decode stage.
Conditional Branches

We do not know whether we have to branch until EX. Inst 5 & 6 are already fetched

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inst 3</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BEQ n</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Inst 5</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Inst 6</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>..</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Inst n</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

If condition is true, we must mark Inst 5 & 6 as unwanted and ignore them as they go down the pipeline. 2 wasted cycles now
Deeper Pipelines

• ‘Bubbles’ due to branches are called **Control Hazards**
• They occur when it takes one or more pipeline stages to detect the branch
• The more stages, the less each does
• More likely to take multiple stages
• Longer pipelines usually suffer more degradation from control hazards
Branch Prediction

• In most programs many branch instructions are executed many times
  – E.g. loops, functions

• What if, when a branch is executed
  – We **take note** of its address
  – We **take note** of the target address
  – We use this info the next time the branch is fetched
Branch Target Buffer

• We could do this with some sort of (small) cache

<table>
<thead>
<tr>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Branch Address</td>
<td>Target Address</td>
</tr>
</tbody>
</table>

• As we fetch the branch we check the BTB
• If a valid entry in BTB, we use its target to fetch next instruction (rather than the PC)
Branch Target Buffer

- For unconditional branches we always get it right

- For conditional branches it depends on the probability of repeating the target
  - E.g. a ‘for’ loop which jumps back many times we will get it right most of the time (only first and last time will mispredict)

- But it is only a prediction, if we get it wrong we pay a penalty (bubbles)
Outline Implementation
Other Branch Prediction

• BTB is simple to understand
  – But expensive to implement
  – And it just uses the last branch to predict

• In practice, prediction accuracy depends on
  – More history (several previous branches)
  – Context (how did we get to this branch)

• Real-world branch predictors are more complex and vital to performance for deep pipelines
Benefits of Branch Prediction

**Without Branch Prediction**

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inst a</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BEQ n</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Inst c</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Inst d</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>..</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Inst n</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The comparison is not done until 3rd stage, so 2 instructions have been issued and need to be eliminated from the pipeline and we have wasted 2 cycles.

**With Branch Prediction**

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inst a</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BEQ n</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Inst c</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Inst d</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>..</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Inst n</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

If we predict that next instruction will be ‘n’ then we pay no penalty.
Consider a simple program with two nested loops as the following:

```java
while (true) {
    for (i=0; i<x; i++) {
        do_stuff
    }
}
```

With the following assumptions:
- `do_stuff` has 20 instructions that can be executed ideally in the pipeline.
- The overhead for control hazards is 3-cycles, regardless of the branch being static or conditional.
- Each of the two loops can be translated into a single branch instruction.

Calculate the instructions-per-cycle that can be achieved for different values of `x` (2, 4, 10, 100):

a) Without branch prediction.

b) With a simple branch prediction policy - do the same as the last time.
Control Hazard Example
Data Hazards
Data Hazards

• Pipeline can cause other problems
• Consider the following instructions:

  ADD R1,R2,R3
  MUL R0,R1,R1

• The ADD produces a value in R1
• The MUL uses R1 as input
• There is a data dependency between them
The new value of R1 has not been updated in the register bank
MUL would be reading an outdated value!!
The Data is not Ready

• At the end of the ID cycle, MUL instruction should have selected value in R1 to put into buffer at input to EX stage

• But the correct value for R1 from ADD instruction is being put into the buffer at the output of EX stage at this time

• It will not get to the input of WB until one cycle later – then probably another cycle to write into register bank
Dealing with data dependencies (I)

- Detect dependencies in HW and **hold instructions** in ID stage until data is ready, i.e. **pipeline stall**
  - Bubbles and wasted cycles again

<table>
<thead>
<tr>
<th>Clock Cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>ADD R1,R2,R3</td>
</tr>
<tr>
<td>MUL R0,R1,R1</td>
</tr>
</tbody>
</table>

- Data is produced here
- R1 is written back here
- MUL can not proceed until here
Dealing with data dependencies (II)

- Use the compiler to try and **reorder instructions**
  - Only works if we can find something useful to do – otherwise insert NOPs – waste

<table>
<thead>
<tr>
<th>Clock Cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>ADD R1,R2,R3</td>
</tr>
<tr>
<td>Instr A / NOP</td>
</tr>
<tr>
<td>Instr B / NOP</td>
</tr>
<tr>
<td>MUL R0,R1,R1</td>
</tr>
</tbody>
</table>

36
Forwarding

• We can add extra data paths for specific cases
  – The output of EX feeds back into the input of EX
  – Sends the data to next instruction

• Control becomes more complex
Why did it Occur?

• Due to the design of our pipeline
• In this case, the result we want is ready one stage ahead (EX) of where it was needed (ID)
  – why wait until it goes down the pipeline?

• But, ...what if we have the sequence
  
  LDR R1,[R2,R3]
  MUL R0,R1,R1

• LDR = load R1 from memory address R2+R3
  – Now the result we want will be ready after MEM stage
Pipeline Sequence for LDR

• Fetch
• Decode and read registers (R2 & R3)
• Execute – add R2+R3 to form address
• Memory access, read from address [R2+R3]
• Now we can write the value into register R1
More Forwarding

- MUL has to wait until LDR finishes MEM
- We need to add extra paths from MEM to EX
- Control becomes even more complex
## Forwarding Example

<table>
<thead>
<tr>
<th></th>
<th>With Forwarding</th>
<th>Without Forwarding</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>LDR r1,A</td>
<td>IF</td>
<td>ID</td>
</tr>
<tr>
<td>LDR r2,B</td>
<td>IF</td>
<td>ID</td>
</tr>
<tr>
<td>ADD r3,r1,r2</td>
<td>IF</td>
<td>ID</td>
</tr>
<tr>
<td>ST r3 C</td>
<td>IF</td>
<td>Stall</td>
</tr>
</tbody>
</table>

**Clock Cycle**

With Forwarding:

- Instruction execution flow optimized by forwarding data from one stage to another, reducing stalls and improving overall throughput.

Without Forwarding:

- Standard pipeline stages with occasional stalls due to data dependencies, leading to increased latency.

Diagram highlights the impact of forwarding on reducing stalls and improving efficiency.
Deeper Pipelines

• As mentioned previously we can go to longer pipelines
  – Do less per pipeline stage
  – Each step takes less time
  – So clock frequency increases
  – But greater penalty for hazards
  – More complex control

• A trade-off between many aspects needs to be made
  – Frequency, power, area,
Data Hazard Example

Consider the following program which implements \( R = A^2 + B^2 \)

```
LD r1, A
MUL r2, r1, r1       -- A^2
LD r3, B
MUL r4, r3, r3       -- B^2
ADD r5, r2, r4       -- A^2 + B^2
ST r5, R
```

a) Draw its dependency diagram
b) Simulate its execution in a basic 5-stage pipeline without forwarding.

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
<th>17</th>
<th>18</th>
<th>19</th>
<th>20</th>
<th>21</th>
<th>22</th>
<th>23</th>
<th>24</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD r1, A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MUL r2, r1, r1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LD r3, B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MUL r4, r3, r3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADD r5, r2, r4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ST r5, R</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

c) Simulate the execution in a 5-stage pipeline with forwarding.
Where Next?

• Despite these difficulties it is possible to build processors which approach 1 instruction per cycle (IPC)

• Given that the computational model is one of serial instruction execution can we do any better than this?
Instruction Level Parallelism
Instruction Level Parallelism (ILP)

• Suppose we have an expression of the form \( x = (a+b) * (c-d) \)

• Assuming \( a, b, c \) & \( d \) are in registers, this might turn into

  ADD  R0, R2, R3
  SUB  R1, R4, R5
  MUL  R0, R0, R1
  STR  R0, x
ILP (cont)

• The MUL has a dependence on the ADD and the SUB, and the STR has a dependence on the MUL

• However, the ADD and SUB are independent

• In theory, we could execute them in parallel, even out of order

  ADD  R0, R2, R3
  SUB  R1, R4, R5
  MUL  R0, R0, R1
  STR  R0, x
The Dependency Graph
a.k.a. Data Flow graph

• We can see this more clearly if we plot it as a dependency graph (or data flow)

![Diagram of dependency graph with nodes ADD, SUB, and MUL, and inputs R2, R3, R4, R5 leading to output X.]

As long as R2, R3, R4 & R5 are available, we can execute the ADD & SUB in parallel.
Amount of ILP?

- This is obviously a very simple example

- However, real programs often have quite a few independent instructions which could be executed in parallel

- Exact number is clearly program dependent but analysis has shown that 4 is common (in parts of the program anyway)
How to Exploit?

• We need to fetch multiple instructions per cycle – wider instruction fetch
• Need to decode multiple instructions per cycle
• But must use common registers – they are logically the same registers
• Need multiple ALUs for execution
• But also access common data cache
Dual Issue Pipeline

• Two instructions can now execute in parallel
• (Potentially) double the execution rate
• Called a ‘Superscalar’ architecture (2-way)
Register & Cache Access

• Note the access rate to both registers & cache will be doubled
• To cope with this we may need a **dual ported** register bank & **dual ported** caches
• This can be done either by duplicating access circuitry or even duplicating whole register & cache structure
Selecting Instructions

• To get the doubled performance out of this structure, we need to have independent instructions.

• We can have a ‘dispatch unit’ in the fetch stage which uses hardware to examine the instruction dependencies and only issue two in parallel if they are independent.
Instruction dependencies

• If we had
  ADD  R1,R1,R0
  MUL  R2,R1,R1
  ADD  R3,R4,R5
  MUL  R6,R3,R3

• Issued in pairs as above
• We would not be able to issue any in parallel because of dependencies
Instruction reorder

• If we examine dependencies and reorder

  | ADD  R1,R1,R0 | ADD  R1,R1,R0 |
  | MUL  R2,R1,R1 | ADD  R3,R4,R5 |
  | ADD  R3,R4,R5 | MUL  R2,R1,R1 |
  | MUL  R6,R3,R3 | MUL  R6,R3,R3 |

• We can now execute pairs in parallel (assuming appropriate forwarding logic)
Example of 2-way Superscalar

Dependency Graph

ADD  R0, R2, R3
SUB  R1, R4, R5
MUL  R0, R0, R1
STR  R0, x
MUL  R2, R3, R4
STR  R2, y

ADD  R2, R3
MUL  R2, R3
ADD  R0, R2, R3
SUB  R1, R4, R5
MUL  R0, R0, R1
STR  R0, x
MUL  R2, R3, R4
STR  R2, y
## 2-way Superscalar Execution

### Scalar Processor

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Stage 1</th>
<th>Stage 2</th>
<th>Stage 3</th>
<th>Stage 4</th>
<th>Stage 5</th>
<th>Stage 6</th>
<th>Stage 7</th>
<th>Stage 8</th>
<th>Stage 9</th>
<th>Stage 10</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD R0, R2, R3</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SUB R1, R4, R5</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MUL R0, R0, R1</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MUL R2, R3, R4</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>STR R0, x</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>STR R2, y</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Superscalar Processor

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Stage 1</th>
<th>Stage 2</th>
<th>Stage 3</th>
<th>Stage 4</th>
<th>Stage 5</th>
<th>Stage 6</th>
<th>Stage 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD R0, R2, R3</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SUB R1, R4, R5</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MUL R0, R0, R1</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MUL R2, R3, R4</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>STR R0, x</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>STR R2, y</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Limits of ILP

• Modern processors are up to 4 way superscalar (but rarely achieve 4x speed)
• Not much beyond this
  – Hardware complexity
  – Limited amounts of ILP in real programs
• Limited ILP not surprising, conventional programs are written assuming a serial execution model
Consider the following program which implements $R = A^2 + B^2 + C^2 + D^2$

```
LD r1, A
MUL r2, r1, r1  -- A^2
LD r3, B
MUL r4, r3, r3  -- B^2
ADD r11, r2, r4  -- A^2 + B^2
LD r5, C
MUL r6, r5, r5  -- C^2
LD r7, D
MUL r8, r7, r7  -- D^2
ADD r12, r6, r8  -- C^2 + D^2
ADD r21, r11, r12  -- A^2 + B^2 + C^2 + D^2
ST r21, R
```

The current code is not really suitable for a superscalar pipeline because of its low instruction-level parallelism.
Superscalar Example

a) Reorder the instructions to exploit superscalar execution. Assume all kinds of forwarding are implemented.
Reordering Instructions
Compiler Optimisation

- Reordering can be done by the compiler
- If compiler can not manage to reorder the instructions, we still need hardware to avoid issuing conflicts (stall)
- But if we could rely on the compiler, we could get rid of expensive checking logic
- This is the principle of VLIW (Very Long Instruction Word)[1]
- Compiler must add NOPs if necessary

compiler limitations

- There are arguments against relying on the compiler
  - Legacy binaries – optimum code tied to a particular hardware configuration
  - ‘Code Bloat’ in VLIW – useless NOPs

- Instead, we can rely on hardware to re-order instructions if necessary
  - Out-of-order processors
  - Complex but effective
Out of Order Processors

- An instruction buffer needs to be added to store all issued instructions
- An dynamic scheduler is in charge of sending non-conflicted instructions to execute
- Memory and register accesses need to be delayed until all older instructions are finished to comply with application semantics
Out of Order Execution

- What changes in an out-of-order processor
  - Instruction Dispatching and Scheduling
  - Memory and register accesses deferred